The future of programming

Posted by Paul Chiusano on Wednesday, December 28, 2011
Labels: haskell, programming, scala

What will programming look like 10 or even 20 years from now? With another new year almost here, now is the time to wax philosophical about the future of our industry. We are on the cusp of a number of major transformations in programming that will make 2011 programming technology, techniques, and ideas seem primitive by comparison. The transformations will occur in several key areas: *tooling and infrastructure*, *languages and type systems*, and *runtime systems*.

Before I get started, let me preface this all with the disclaimer that this should be taken with a grain of salt, in the spirit of being provocative, and these are not really predictions *per se*. There are lots of reasons why inferior technologies might continue to dominate the industry longer than expected. I won't talk much about that here, although it's certainly interesting. No, instead I want to give *my vision* of the future of programming. If there were no accumulated network effects attached to all the mediocre technologies currently in use for programming, if the programming world were given a clean slate and the directive to invent the future, what would I want to result?

An overarching theme to all these transformations is the move away from *incidental structure* and its close cousin *incidental complexity*. Manifested in various forms, these factors are major barriers to the building programs of greater complexity, and many of the major transformations will be to enable lifting of these barriers. The resulting programming world will look much different--orders of magnitude more code reuse, ease of deployment, efficiency, and much more.

Where we begin

The first major change is the move away from storing programs as text, split across "files". You can be forgiven for thinking of this as a minor thing. Who cares how programs are stored or represented, right? In fact, this major source of incidental complexity has spillover effects all the way down the programming toolchain, starting with the pollution of programmers mental thought processes, continuing to development environments (which I'll discuss next), and then to build systems and deployment. Like many of the things I discuss here, the full extent of lost productivity due to this incidental complexity is below most programmers radars. Programmers have unconsciously accepted this lost productivity, to the point that many now rationalize their dependence on files and text for programming.

But what about intentional programming, you ask? Or Smalltalk development environments? The idea of moving away from programs as text spread across files is not exactly new, but the devil is in the details. Previous efforts to rid ourselves of this incidental structure have failed in part because they merely substituted one arbitrary structure for another. For instance, intentional programming advocated the good idea of separating code presentation from its storage, substituting instead another format like XML with only slightly less incidental structure. Any advantage conferred by the new format and structure was not exploited to significant benefit, and there was a huge, very real cost in terms of lost network effects from all the text-based tools that could no longer be leveraged. Likewise for Smalltalk development environments. Also, somewhat ironically for Smalltalk, being an OO language, its fundamental premise includes one of the most arbitrary choices of all, in some sense the ultimate generator of incidental complexity, namely, *the decision of which class should implement a particular method.*

There is a real barrier to entry here for any new technology. It doesn't need to just be better than the

antiquated status quo; it needs to be *significantly so* to overcome the network effects and switching cost advantage that status quo technologies inherently posses. In my opinion, none of the proposed text-in-file alternatives so far have come close to overcoming these handicaps.

But this is nothing fundamental. We should not conclude from these failed experiments that the idea is unsound. It's merely been poorly executed. The first major shift in getting this right is not storing programs as text at all, not even some normalized textual form like XML, and not some normalized binary form which nonetheless contains a huge amount of incidental structure. Likewise, get rid of files. Instead, a codebase is stored relationally, as a database, with some version of datalog being used for querying and update. Careful thought will be given to this query-update language so that many of the large-scale refactorings that are currently difficult or impossible can be fully automated, or guided automated, with just a few lines of this codebase transformation code. As a simple example, we eliminate the notion of where a function or datatype "is located". Instead, each function gets a unique id (perhaps content addressed, to enable a simple, automated form of code deduplication), and a separate `names` table maps these ids to names for purposes of rendering the code in some fashion to a human (and in principle, there is really no reason why there can't be multiple `names` tables, with different programmers choosing different names for the same operation). This representation trivially enables the "renaming" refactoring in just a single table cell update.

Let me sketch out a few additional thoughts on how this could work. First, I am not advocating for datalog *syntax*. I don't care about that. The key functionality enabled by datalog over and above the relational algebra is the ability to express transitive closure and mutual recursion guaranteed to terminate. Together these features enable many of the common queries we'd like to express in transforming and querying our codebases. For instance, here is a hypothetical query to find all references to a given function id, `fid`. Don't worry if the syntax looks alien or doesn't make sense. The key is more that this query is just a few lines of code to express, and it can be reused and built upon.

```
-- propagate reference to containing apply
refs(Id) :- apps(Id, fid, _).
refs(Id) :- apps(Id, _, fid).
refs(Id) :- refs(X), apps(Id,X,_).
refs(Id) :- refs(X), apps(Id,_,X).
-- any lambda whose body is or contains fid
-- is considered to reference fid
return(Id) :- lambdas(Id,_,Id1), refs(Id1).
return(Id) :- lambdas(Id,_,fid).
```

Current IDEs, with their enormous development staff and reams of special purpose code, support a very limited subset of code transformation/querying operations, poorly, slowly, and with huge overhead. As a result, there is an entire subculture of programmers (myself included) who have largely rejected IDEs in favor of minimal, less feature-rich but more dependable text editors. Future development environments will take the refactorings now supported by the most advanced IDEs as the most insignificant starting point, and build from there.

Large scale refactorings are the inevitable consequence of achieving the reuse needed to build programs of significant complexity that don't die of heat death. Refactoring times in this new model will go from weeks or months to hours, and writing code to transform a codebase will become a separate but critical skill, distinct from the usual act of programming. That is, programmers do not simply conceive of a refactoring (which is often quite simple to express to another programmer), then begin a tedious, manual and error-prone process of text munging to implement it. Instead, the programmer conceives of a refactoring, then conceives of a code transforming program to implement the refactoring, then applies this transformation to the code database, all in the span of a few hours.

The code as database concept has spillover simplification effects in other areas of tooling, in particular version control. The problem of handling merges of ordered sequences of characters spread across files and directories with yet more arbitrary structure is extremely difficult, resulting in a huge amount of complexity in the area of version control software. The difficulties have led many to settle for what are arguably inferior VCS models (Git, Mercurial), where changesets form a total order and cherrypicking *just doesn't work*. In the future, with code databases, we'll see a resurrection of Darcs' patch theory, only this time, it will work *exactly as expected*, and will be trivial to implement.

Related to this notion of code databases, we get much more fine-grained dependency management. Dependency management is another serious problem for software development, a huge hindrance to code reuse, and once again, something that is simply below many programmer radars. A library will often introduce some new interface or abstraction and give several instances of that abstraction for particular concrete types. The library now *depends on* these concrete types being available for compilation. Including this library now means pulling in all these dependencies, even if you just require the one instance. The overhead of doing this in conjunction with the existing complexity of builds (again partially caused by the programs-as-text-in-files paradigm) means code is reused much less than would be possible or easy otherwise.

In the future, we'll be able to conditionally specify code, and not using an ad hoc, 1970s technology like the C preprocessor. The datalog query and update language will factor in here, and allow us to express things like: if a concrete type `X` is available in the codebase, define some set of functions and new datatypes. Likewise, dependencies will in general not be on "a library" or "a module". A function will depend only on the set of functions and types it references, and a codebase will be a much more fluid thing. We can extract any connected component of functions and datatypes from a codebase, and this is trivial, automated transformation supported by the query language. There are some unknowns here, all solvable, around versioning, and they are related to how we reconceptualize version control as a partial order of changesets, with no extraneous dependencies due to the representation of programs as ordered sequences of characters.

Code editing, IDEs, and type systems

Code editing will be done in structural editors, which will look nothing like the existing batch of IDEs that are little more than glorified text editors (and they are actually rather poor text editors). In a structural editor, the programmer will construct expressions which may have holes in them not yet filled with terms. Importantly, these structural editors will be type-directed, so for any given hole the programmer can be presented with set of values matching the expected type, ordered in some sensible way. The editor will perform local program search to enable autocompleting of multiple expressions. If you've ever seen someone live-code Agda, you'll know how powerful and productive this idea could be. Yeah, the actual interface for programming Agda is still kind of 1970s (a custom Emacs mode), but the idea of type-directed editors is powerful. It makes it clear that types are effective not just at preventing many software errors, but also in *guiding development*. They are a powerful tool to augment our puny programming brains.

Along these lines, the rise of type-directed development in languages with real, state of the art type systems will mark the beginning of the end for dynamically typed (aka, single-typed) languages. Dynamic typing will come to be perceived as a quaint, bizarre evolutionary dead-end in the history of programming. This is already widely accepted in some programming circles, where the general consensus is that most dynamic typing advocates are not familiar with type-directed development in a language with a real type system and are basically unaware of the state of the art in type systems and programming language theory. With some additional developments in type systems we'll see any last of any advantages to dynamic typing completely evaporate.

A related transition is that types will become even more critical and type systems will grow features to better handle data normalization, the lack of which is a major source of incidental structure and complexity. A big problem in large codebases is data normalization. Lack of data representation normalization is responsible for the significant amounts of plumbing code involved in aligning two pieces of code so they can talk to each other. Most of the mainstream programming world isn't really aware of this issue because code reuse in most imperative codebases is extremely limited (because side-effects don't compose well).

In the functional programming world, there is insane amounts of code reuse happening, but along with this comes a fair bit of plumbing. As a small example, consider a function, `f`, of type `X -> Y -> Int`, and a value, `xs` of type `[(Y, X)]` and suppose you want to apply `f` to the list. An experienced functional programmer will bust out `map (uncurry f . swap) xs` in about 3 seconds and not think twice about it. But this code is total boilerplate, there to convince the compiler this is what you really want to do, and is noise when reading the code. Yes, you are acheiving reuse (an imperative programmer would still be writing out a `for` loop for the millionth time instead of reusing higher-order functions) but it should be cleaner. If this code needs to exist (and I'm not sure it even does, see below), I'd rather get to this point in the editing process and then tell my editor to map `f` over `xs`. The editor will search for a program to make the types align, show me the program for confirmation if I request it, and then *not* show this subexpression in the main editor view, perhaps just showing `map f* xs`, where the * can be clicked and expanded to see the full program.

Even better, perhaps we can get away with much less plumbing in the first place. Plumbing can be eliminated from code in two general ways--the first, which I've just discussed, is to have plumbing code written for you automatically, then hidden unless requested. The second is to have more normalized types and type systems so that there *is no incidental structure for code to broker between in the first place*. To support this we'll see things like row types, unordered tuples, type sets, and so on, and any related type system machinery needed to make all this feasible *and convenient*. An idea I'm interested is explicit separation between the model underlying a type (which could be something very normalized, with no incidental structure), and views of that type, which may contain some additional structure. Functions will generally be written to operate over the model, from which any number of views can be reconstructed post-transformation. The compiler or runtime is generally smart about choosing runtime representations to avoid unnecessary round trip conversions between different representations.

Language runtimes

Non-strict languages will come to dominate the programming world, due to the increase in code reuse and modularity that comes with pervasive non-strictness. As I've argued before, optional laziness doesn't cut it. As with other issues I've mentioned, the problems with strict as default aren't apparent to most programmers, even those who view themselves as well-versed in functional programming. The problems with strictness only become fully apparent as you get much further along in the development of the FP style, in particular, after the discovery of combinator libraries and the further abstractions that develop to remove duplication across these libraries. This has been a source of tension in the Scalaz library, which supports functional programming in Scala, a strict-by-default language, and also in our Scala codebase at work.

The only real problem with laziness has to do with reasoning about space usage performance, and evaluation stack usage. These problems get a lot of play among people who enjoy rationalizing their ignorance of Haskell and FP in general, but the truth is some additional research and smarter evaluation strategies can address the problems. There's nothing fundamental here suggesting we should throw up

our hands and resort to strict evaluation.

The usual normal order evaluation is guaranteed to terminate if any is, but reasoning about is space usage in this evaluation order is problematic. We can do much better. Besides static strictness analysis, which only covers a fraction of the cases we'd like, we can conceive of additional evaluation orders which terminate for the same set of programs as normal-order evaluation, and which can propagate additional strictness information. The one I am particularly interested in I refer to as *specializing, strictness-propagating evaluation*. I'll elaborate on this in another post, but in this evaluation model, calling a function is something like two communicating coroutines. When calling a function, the callee begins evaluating its body, yielding control back to the caller when it needs its first argument, and also indicating whether that argument should be strict or lazily passed, using whatever information is available at runtime. Subsequent arguments work similarly. As a result, functions are automatically specialized as arguments are passed, and we do not construct thunks if they are going to be consumed strictly by a subsequent callee. This can be implemented efficiently using just two call stacks, and there are various optimizations to the scheme. It is intended to augment, not replace, the existing static strictness analysis and argument passing.

In general, the goal of new evaluation strategies should not be efficiency (though that is a nice goal as well), but simple reasoning about space and evaluation stack usage, so that these things do not depend as they currently do on incidental details of how a function is factored or what the compiler chooses to inline (being forced to reason about these things in Haskell breaks the black box abstraction of functions that FP and higher-order programming depend on).

Language runtimes and VMs will grow to support these new evaluation strategies, and the current crop of "general-purpose" VMs that are actually quite specialized for strict, imperative languages (the JVM, the CLR) will likely die off.

Code distribution and the future of the web

What of the web, javascript, html 5 and beyond? Are we going to keep hobbling along with these technologies indefinitely? I don't think it's too controversial to say that writing applications of any significant complexity as a *web application* involves far more work than would be required with access to a truly powerful client-side language. Again, I don't think many web programmers realize just how bad the situation is. In comparison to mainstream languages like Java, C#, or Python, Javascript isn't so bad; in some ways it's even a better language. But compared to a programming language with a good type system and real support for FP like Haskell, Scala, and whatever the next generation of languages can bring, Javascript is quite sad.

Of course, proponents of existing web technologies are always finding ways to rationalize the status quo, pointing out how much better things are today than they used to be. That's maybe true, but why settle?

I envision a future where Javascript dies off, as do browser-specific plugins like Flash, and instead we'll see client code written in arbitrary, compiled languages, using something like NaCl. This will be combined with a signed code caching and dependency tracking mechanism, so that for instance, you can distribute an application that downloads the entire Java virtual machine and other dependencies, but only if they aren't already cached on your local machine (and the signatures match, of course). This changes how we think about software. Software won't be something you "download onto our computer and then run". Instead, software exists "out there", and you run it. As an implementation detail of running it, it may choose to download some additional code it needs to do its work.

This will end the monopoly that html+javascript has on client-side web applications and largely

eliminate the switching costs of moving to new client-side technologies. A few lower-level protocols and standards will be enough to tie everything together and maintain the good parts of the web today (I'll say more about this next), but nothing so high-level as specifying the client side language for interaction (Javascript, or Dart, or whatever) or the client-side language for layout and display (html + CSS). In the future, client-side code is written in whatever language, compiled to native code if desired.

In place or in addition to native client support, another option would be some sort of in-browser low-level VM designed by people who know what they're doing, who are fluent in the state of the art in programming languages and runtimes. In other words, not the people who designed Dart, Go, Ceylon, or any of the other recent languages that are 30 years or more behind state of the art. We need something that actually supports languages of the future, not something that rearranges the deck chairs on the titanic sinking ship that form the current batch of mainstream languages.

What of the suggestion that we simply use Javascript as the "assembly language" of the web, use it as a compilation target, and avoid programming in it directly? This obviously is not ideal. Compiling a real programming language like Haskell or whatever is next through Javascript or Dart is obviously not how anyone would not how design things today given a clean slate. Even if this were possible to do well without bloating client code size beyond what's acceptable, there is the inevitable efficiency hit of compiling to a language whose performance is as bad as Javascript. When you think about it, it makes no real sense to be reverting to 1/10th or 1/100th the speed of what native code or a well-JIT'd VM could provide, *purely for ease of deployment*. We don't need to be making this tradeoff, and with the rise of something like NaCl and/or a real browser VM, we won't have to.

There's one wrinkle. There are tremendous network effects on the web. This is part of its power, and its usefulness, and we don't want to lose it. We do still need the moral equivalent of urls and hyperlinking but one thing we don't necessarily need is the DOM. What will take its place? Don't we need the DOM to enable mashups and services like search engines? Actually, no. DOM munging and traversal is not the only way programs could obtain information from applications on the web, and when you think about it, it's actually rather primitive. Why screen scrape when you can call a real API? This transformation is already sort of happening; most modern web applications expose APIs in the form of REST+JSON. It just needs to be taken a little further.

What would this look like? Well, we would need a standard type system for the web. By this I mean a standard way for the applications that live at a url to expose a module of types and functions they support. The underlying type system would be expressive enough to encode richer data than what JSON currently provides (which besides being dynamically typed, cannot even represent sum types effectively) and will support for algebraic data types and some of the type system features alluded to earlier. With this in place, standards will arise for certain function signatures, with standard meanings attached to them. So, for instance, rather than the googlebot crawling the DOM looking for links, it invokes the `getAllLinks` function of the application at the given url, which returns an actual list of url values. `getAllLinks` is some separate standard, and new ones can arise on an ad hoc basis, as we grow new modes of interaction with web sites. There will be certain near-universal standards (like `getAllLinks`) and more specialized ones, specific to the web application in question (for instance, Facebook exports certain functions and datatypes that are specific to Facebook, which are unlikely to be implemented by other sites, though this is certainly not a requirement).

Already, we can see this happening somewhat: there are various ad hoc APIs and mechanisms for basically controlling how web crawlers should interpret the DOM.

With a standard way of interacting with web sites programmatically, there's no longer any need for the DOM and we can see a proliferation of innovation of different display and layout technologies.

Closing thoughts: the rise of functional programming

I've hinted at this throughout: functional programming will *absolutely win*. Not everyone shares my views, but many of the improvements I've talked about start with the assumption that we are working in a functional, post-imperative world. FP provides a dramatic increase in productivity due to massive increases in code reuse and the ease of reasoning about functional code (not to mention ease of parallelization). The industry is slowly starting to understand this, but it doesn't really matter if many programmers are still for whatever reason resistant to learning FP (which is unfortunately true). Eventually, the selection pressures of a competitive market will weed out less productive and effective techniques, and this largely means the death of imperative programming as we know it, except in certain niche areas. Regardless of initial biases or attitudes, programmers and companies who wish to stay competitive will be forced to employ FP.

As for why FP hasn't gained more prominence already, well, perhaps I'll write more about that in another post. What I will say is FP is currently reaching a tipping point enabling its wider adoption and relevance. There are several factors in play: functional language compiler technology is advanced enough, computers are fast enough, and most importantly, the FP community is rapidly discovering all the necessary techniques to organize large programs that preserve referential transparency. The Haskell community has mostly led this charge, Haskell being the only widely-used language that, due to its non-strict evaluation, had no choice but to fully commit to preserving referential transparency throughout. Ten years ago, admittedly, there was a good chance that expressing some programs purely functionally involved what was basically new research. Today, many of the required techniques are known, and expressing even the most imperative-seeming program functionally is possible, and for an experienced functional programmer, pretty effortless. There are still interesting open questions of how to express certain programs, but these are diminishing rapidly.

That said, I understand why many people claim FP is too hard, or it's unnatural or awkward, etc. Like many worthwhile subjects, attaining fluency in FP is difficult and requires dedication. Once this level of fluency is reached, though, expressing functional programs is quite natural and effortless (of course, software design is still hard, but finding functional designs becomes easy with practice). For people looking in, the techniques of FP seem opaque and unnecessarily difficult. For people on the inside, there's nothing difficult or complex about it and the benefits are enormous. For those who would criticise FP, I think a little humility is in order. To draw an analogy, no one without mathematical background would feel equipped to dismiss or criticise an entire branch of mathematics ("real analysis is a stupid idea"), and yet programmers with barely a cursory understanding of FP regularly (and loudly) criticise it.

I see why some people are frustrated with some of the existing resources for learning the subject. But it's wrong to dismiss the subject on that basis, or on the basis of personalities of people (myself included!) who feel that FP is more productive; objectively, either FP is worth knowing because of the productivity and other benefits it provides, or it isn't. For my part, I am co-authoring a book on FP that I hope is helpful to some who are interested in learning the subject. With the solidification of FP techniques I expect to see more and more resources like this, to the point that fluency and the huge resulting benefits are something that any motivated programmer can work toward, without encountering some of the discouraging hurdles to learning FP that are present today.

Will the future of programming look anything like what I've laid out here? Maybe, maybe not. But I certainly hope it will.

45 comments:

Derek Elkins said...

The popular view of the "performance problems" of laziness do not have so sinister a source. It basically comes down to this: I program Haskell pretending that it's an eager language and when things perform poorly, it's laziness's fault. Basically, it's mostly a problem of ignorance and lack of education.

The annoying thing is the blithe double standard. If I were to treat Scheme like it was a lazy language, and write the non-tail recursive version of map, when it stack overflows it's not eagerness's fault, it's my fault. When I write the tail recursive version of sum in Haskell and it stack oveflows, it's laziness's fault.

Part of the reason for this attitude is that most of the people learning Haskell, or at least most of the loud ones, are experienced programmers who think their experience implies that they know how to write reasonably fast code. They don't seem to factor in that none of their experience included laziness.

December 28, 2011 11:02 PM
Steven Shaw said...

s/swap/flip

December 29, 2011 12:06 AM
Wojciech Soczyński said...

You have some interesting ideas in this post. But I think that the real rise of functional programming to a dominant paradigm will never happen. Why ? Because most of the programmers are the bad 80% from the Pareto rule and they just too stupid/lazy to master it.

Just look at the now dominant OOP paradigm. People claim, that they write object oriented code and instead they writing procedural code using OOP artifacts (classes, methods, inheritance etc). Not to mention, that most of them, don't get the difference between object orientation and class orientation.

In my opinion, there will be more interest in FP and we will see more FP features in the common languages (Java/C#/PHP etc). But I would not expect a major mindshift.

December 29, 2011 1:19 AM
Anonymous said...

Quite interesting that you chose to represent your "refactoring program" in a textual manner.

We see this idea of using non-text representations every once in a while and it hasn't taken off in any significant way -- not even in niche applications (at least AFAIA). Maybe it really is just a bad idea?

December 29, 2011 2:11 AM
Jesper Nordenberg said...

While I agree that the ideas you present are compelling, but I don't see most of them becoming

commonly used in 10, or even 20, years from now (I hope I'm wrong :)). Just look at what has happened in programming language field the last 20 years: C++ is still alive and kicking, the Simula-based (invented in the 60-ties!) languages Java and C# are the dominant statically types languages, compilation and source editing are basically unchanged. I only see two major advances during the last 20 years: the move to running on virtual machines, the rise of IDE's incorporating tools based on program analysis. As you write we're starting to see a new major advancement towards functional programming in mainstream languages (even Java will get lambda expressions soon ;)). In 20 years I would be very surprised if we don't still edit our programs in text editors, still use dynamically typed languages and OO is still commonly used (although FP will be be much more commonly used than today).

December 29, 2011 3:24 AM

Anonymous said...

I think the opposite will happen and static typing will die off for most use cases.

December 29, 2011 5:28 AM

Paul Chiusano said...

@Derek - that is definitely true. But I do think reasoning about performance in lazy languages can get even easier than it already is, and we should strive to make this happen rather than accepting the status quo as good enough. That said, I don't think any of this should dissuade anyone from using Haskell as it exists today.

@Steven - it actually works as is. :) The pair is being swapped (a,b) -> (b,a), then passed to the uncurried f, which accepts tuples. Though it might be a bit more efficient to just write it as uncurry (flip f).

@Wojciech - while I agree that most people who supposedly practice OOP don't really leverage it I would still say that OO is the dominant paradigm. The same could be true for FP. Also, FP is a bit more clearcut - either you are programming with pure functions, or you aren't.

@Jesper - Well, like I said, there are lots of reasons why inferior technologies might stick around longer than expected. :) I can't predict that sort of thing. All I'll say is that I'll be pretty disappointed if in 20 years we are still roughly where we are now, using the same tools, the same languages, and the same paradigms!

December 29, 2011 5:28 AM

Anonymous said...

Recommended reading if this kind of thing interests you:
http://www.vpri.org/pdf/tr2011004_steps11.pdf

December 29, 2011 6:17 AM

TechNeilogy said...

I agree with most of what Wojciech Soczyński said, but I would temper it with a ray of hope: there are external forces driving the adoption of FP that make it harder to "fake" the adoption of

FP the way OOP is often counterfited. These external forces include, but are not limited to: concurrency, distributed programming, fault tolerance, the need to prove correctness, etc.

December 29, 2011 6:46 AM
Viktor said...

Great post. I do not feel competent enough to either refute or advocate the ideas presented here but in my opinion one thing is clear: this kind of forward thinking is extremely valuable and people need to find some way to experiment with such ideas and try to implement them. Otherwise we'll continue writing the same code for decades, like we've been doing so far.

The sad truth is that no one wants an idea, they want a working product and don't support any efforts until those efforts produce something of tangible value, which makes as much sense as "colorless green ideas sleep furiously".

December 29, 2011 6:53 AM
Wojciech Soczyński said...

Techneilogy - I agree with your point about the forces driving the adoption.

But we must also notice, that if someone writes in a hybrid language, there is always some place for the violation of FP rules. Just to mention Scala - despite it has great FP features, you can still write it in the old procedural way.

December 29, 2011 7:05 AM
Senthil KUmar B said...

In few years time , you might even see programming with out programmers :)

December 29, 2011 7:07 AM
Unknown said...

One key problem will be the recognition that we are
generating a lot of dead legacy code. See Sourceforge.
Technology that allows code to "live" beyond the
involvement of the original authors will become
important.

Our current technology of trees of little files causes
us to discard the most important information. Today's programmers would strip the equations
from a calculus textbook, put them in directories
organized by chapter, and throw away the text.

I believe that Literate Programming (Knuth) will
become more widespread. Imagine being able to
hire someone new, give them your literate program
and send them for a 2 week paid trip to Hawaii.
When they return they are able to maintain and

modify the code as well as the current team.
The Hawaii test is the key criteria to measure
whether your literate program is successful.

See "Lisp in Small Pieces" for a great example.
See "http://daly.axiom-developer.org/litprog.html"
for an example using HTML.

Tim Daly

December 29, 2011 7:26 AM

Anonymous said...

I really do not believe FP will be mainstream any time soon. While the FP concepts there are clear and beautiful, it will not happen until someone will invent the "Python" of FP. Right now I guess FP is at the "Tcl" phase.

Also, I do like the idea of filling the database instead of writing those pesky text files, but I am not sure it will be real fun, which (widely understood) programming gives.

I had similar thoughts that managing software applications should be additive: pour two code databases together and you get twice the features of the originals. And I guess the semantic web concept is exactly about that. That is, I think, in data-heavy applications it would fly nicely. Not so sure about logic-heavy applications though. How to reuse code from, say, floating point division in regular expression engine? Even if we have central repository for all the algebra formalized as nice data types and functions up to date?

There is certainly something psychological in the proliferation of today technologies. And the truth may be even greater than shift towards FP.

December 29, 2011 11:13 AM

Barry Kelly said...

I think you're completely off base in almost every respect.

Functional programming will not win in the arenas where most code is written: business logic. That's somewhat painful to say, because FP approaches - in particular, functional reactive / dataflow programming - can reduce complexity of specification, and in particular can reduce duplication by specifying behaviour declaratively in terms of underlying state, rather than imperatively at the point of each state change.

The problem, rather, is that everyone up and down the line describes the desired behaviours in imperative terms, and it takes some thinking to compress these imperative descriptions into declarative statements. That thinking needs to be taught; but the people who code these systems frequently don't even start from a computer science background, and when they do, they're the types who found functional programming hard to understand and pointless. It's not because they're not smart enough (Wojceich is deeply deluded if he really believes this); it's because they've put their smarts in other areas, like understanding the business, and FWIW, will probably make more money that way too.

Lazy vs eager: I personally don't have a strong preference for either for a functional language; perhaps I lean slightly towards lazy. But since I don't think FP will win **everywhere**, I don't think it's going to be **that** interesting a question; imperative programming with lazy evaluation is a quick recipe for insanity.

Moving away from a text-first representation of code: not going to happen before computers write our code for us, and human programmers become obsolete. Text is too flexible, and the alternatives too structured. Anyone who chooses to put their code into such a proprietary format (and it will be proprietary, one way or another) will be to blame for all the pain giving up such leverage inflicts on them. If anything, the trend is away from tight structure and syntax, and more free-form mixing of program text with other text.

FP will probably succeed massively in a few niche areas. Most likely infrastructural areas, areas that are intrinsically functional (e.g. server request / response). But even then, the domain probably most suited to functional programming, compilers themselves, functional programming hasn't yet been a commercial success.

December 29, 2011 12:59 PM

Anonymous said...

This post should be titled "how people with IQs of 160+ will code in ten years"

Code in a database? NoSQL and a host of other libraries are written because even above-average programmers dislike SQL.

Functors? Closures? Thunks? Haskell? Forget it.

The sad fact of smart people programming is that no matter how "powerful" your tools, the real world products get dwarfed by armies of people coding in "primitive" languages.

Haskell is just an ivory tower language. It's for really smart people to sit in ivory towers and chuckle at other people.

More to this point, you haven't really said anything about the real gains in the last decade: programmer interaction and code sharing. Your entire blog post is about tools that smart people will use to write completely inscrutable, unshareable code as quickly as possible.

Sure you can outcode 10 people, and maybe even 20, but never 1000s.

Sorry, code as database is interesting, but there's a reason LISP has languished for decades. Plain, obvious reasons that really high IQ people, completely in character with their stereotypes, can't even begin to understand.

December 29, 2011 2:02 PM

jed said...

There is already a textless representation largely pure functional programming language, in use by non-self identifying programmers in businesses all over the world – the spreadsheet!

December 29, 2011 2:04 PM

Paul Chiusano said...

@Barry - Just to play devil's advocate, I'm not sure it's so inevitable that amateur programmers think imperatively. If (functional) programming were taught in school and given similar weight as other subjects like math, reading, etc, perhaps any educated person could find functional thinking quite natural. I've heard interesting anecdotal stories about how easy FP is to teach to people, even children, with little or no prior exposure to programming. If you don't have to unlearn imperative ways of thinking, maybe it's not so bad. What makes you so sure the "imperative" mindset is so fundamental? Granted it might take a while for all this to trickle down...

That said, sure, I wouldn't necessarily make a prediction that imperative programming will be totally wiped out among all programmers, professional or amateur. But just like in other fields, the professionals might use different tools and techniques than your average amateur practitioner. (Like I said, I think there will continue to be a niche for imperative programming, but it won't be the dominant paradigm anymore, and most imperative programming may be done within the confines of an FP language - in many ways, Haskell already supports imperative programming quite well)

Text is fine as an entry mode for programs and I think text entry can be integrated nicely into structural editors. You enter your programs as freeform text, which get slurped up and placed in the database, which can even prettyprint it back to you as text. The point is that codebases will have a standard normalized relational form suitable for expressing large-scale querying and transformation beyond the current ad hoc set of tools that current IDEs provide.

Beyond program entry and "localized" editing, the only advantage text has going for it is there are a lot of tools for manipulating it. The problem is these tools don't work well enough for the kinds of things we want to do with code. For doing actual refactoring on code, and for doing version control, text-based tools are pretty awful IMO, at least compared to what could be built. Yeah, there are network effects / switching costs to overcome for this to get adopted, but if the benefits are big enough, it could happen!

December 29, 2011 2:41 PM

Zoheb said...

I think we should have programming by example pretty soon. Some on-going research on program synthesis is also worth watching out for. Rather than specifying the type of a program and have the IDE search for a target value, we specify sample input/output values from which the IDE infers the type of the function and suggests formulae that satisfy not just the type but also the input/out examples specified.

December 29, 2011 2:44 PM

Anonymous said...

Others have touched on it, but something huge has happened in the last 10 years. Dumb, idiotproof Java and Google have enabled hundreds of thousands of people to work as programmers. Few of them truly understand what they are doing but they can search, cut'n'paste, bash it until it compiles and then claim a salary.

Luckily, for us, there is a higher ground that includes Haskell that makes us many times more productive (particularly if you include defect resolution) but this will always be a niche area until businesses recognise the value of skilled programmers and don't just assume that most Indians at $100 / day are just as productive as a skilled programmer at $750 / day.

December 29, 2011 2:52 PM
Paul Chiusano said...

@Anonymous - are you so sure it's an intelligence thing? Because I don't think so. FP is just different, and in many many ways much easier than imperative programming. It does mean accepting certain constraints on your programs, but as the techniques needed to express programs given these constraints become known, you don't need to figure them out or think really hard every time you go to express some program. Ten years ago, that was not the case. Five years from now, the patterns will all be known and be "so simple even a caveman could do it". And there will be much better resources for learning FP as a result.

If FP seems like an ivory tower now vs other paradigms, I think it's more because FP is still not totally mainstream, and anyone who knows it is probably interested, curious, and motivated. This gives them a leg up on anyone who only learns something if it is obviously, directly required for their day to day work, and only learn on the job, never on their own time. Again though, I'm dubious that it's actually an intelligence thing. I think it's more a question of being interested, motivated, and curious. Sure, some base level of intelligence is required, but beyond that it's more about these other factors.

December 29, 2011 3:00 PM
Anonymous said...

What you describe has been largely implemented by Model Based Software Engineering tools. In some industries, like avionics and telecoms it is the dominant methodology. And these tools have been evolving for over 30 years. Of course, since it is such a huge competitive advantage the users do not advertise their order of magnitude improvements in productivity and quality.

December 29, 2011 5:55 PM
Anonymous said...

the truth is some additional research and smarter evaluation strategies can address the problems. There's nothing fundamental here suggesting we should throw up our hands and resort to strict evaluation.

I read that as "There are problems with laziness that haven't been solved yet, but may be someday." That's...a tough sell.

December 29, 2011 8:48 PM
Paul Chiusano said...

@Anonymous - regarding laziness, let me be clear that I would still much rather use a lazy language than a strict one. Also, like Derek said in an earlier comment, I think the problems with

laziness are overblown, mostly by people who don't have much experience programming in a lazy language and expect it to work just like a strict one. That said, I think the situation can be improved further. But let's not throw the baby out with the bathwater!

December 29, 2011 9:14 PM

Anonymous said...

This is simply elitist rubbish. FP is for language snobs and Dynamic languages are for people who don't have to maintain large code bases.

In the real world, OO languages that cherry-pick good ideas from FP (think Scala, but done better) will win out.

December 29, 2011 11:14 PM

Wojciech Soczyński said...

@Barry Kelly: you are right, that FP thinking should be taught, because people think in imperative terms.

Moreover, I think that programming languages influence our thinking, because they naturally impose some idioms to our mindset.

This is just as it happens with natural languages - someone who speaks native English, view the world completely different that someone who speaks native Chinese.

But I also think that you are wrong when you are speaking of programmers putting their smarts in other areas - especially understanding business. From my experience - 90% of programmers see the code in purely technical terms. They don't connect real life models to code artifacts.

They approach the code in the CRUD way, for example - they see a process of a user buying some thing in an e-shop as a database operation rather than a business process.

You are lucky if you have the pleasure to work with programmers that understands business ;)

December 30, 2011 12:42 AM

Anonymous said...

It is a feature I'm definitely looking forward too!

December 30, 2011 5:35 AM

Anonymous said...

great writeup but a tad strong on the prognostics side. as for functional programing check out mythryl http://mythryl.org/

December 30, 2011 8:43 AM

Greg Wilson said...

I think SCID (source code in database) and FP are orthogonal: the success or failure of one will have nothing to do with the s|f of the other. cf. http://queue.acm.org/detail.cfm?id=1039534 (from 2004)

December 30, 2011 11:43 AM

Anonymous said...

when you put your code in a db you probably still have a programmer friendly syntax to interface with this database?
many ides already parse the source files and manage the ast pretty much like in an in-memory database for complex refactorings.
i think even more interesting will be ide frameworks like eclipse xtext, that enable easy source based code transformations.
for different code versions and flexible dependencies i would recommend a look at the osgi project.

December 30, 2011 4:51 PM

Philippe Monnet said...

Interesting read but I think you glossed over a bit to fast on Smalltalk when talking about structuring code away from text files. Ironically Smalltalk was my first love over 20 years ago. The concept of "image" (essentially a whole suite of Object instances and associated code in memory) only required one file and could be just saved after updates. And the ability to query code, find implementors and references to a given class or method was ahead of its time. What limited its adaption was the lack (at the time) of packaging tools.

I am not convinced that FP will take over. Instead I think that more developers will intermix OO and FP more and more.

But at the same time I think Javascript will eat at the language usage pie as its platform is ubiquitous and its learning curve minimal. As it evolves, it might gain more and more respect.

December 31, 2011 10:34 AM

gasche said...

Reading your post made me angry.

1. There are interesting bits in it. I found an idea inside that was *new* to me : normalizing types by isomorphisms to avoid boilerplate. I don't think that's the right way to avoid boilerplate, but it's thought-provoking anyway. In all cases I expect most reader to find something interested and new in this post, which is good.

2. It is incredibly contemptuous and arrogant in the details. The local structure of your post is " is mediocre, here is that will magically solve ". Sadly, the state of the art that you're criticizing is in a lot of cases the kind of things *you* (or someone like you) would do if you wanted to *make concrete progress* on the idea you express. I have a sickening thought of yourself starting to work on some of these topics, going further that the mere one-paragraph draft, investing painful efforts into getting people to use your improved solution, and a different copy of your self laughing at this "mediocre" undertaking and claiming that some half-thought idea (which is

essentially the same thing, without the details, work and sweat) is obviously better.

3. It is the kind of things that I could write or have written. I could see myself developing those ideas (or related ones), being equally arrogant, making the same mistakes; only I wouldn't see them, because it's easier to judge the others.

Here is how you could make me less angry with your next post (I just subscribed to your syndication feed, which I should have done long ago):

- Concentrate on the ideas

- Avoid being a douche about the related works. For a start, you could try to avoid the words "mediocre", "don't know what they are doing", "failed", "poorly", "arguably inferior", "1970", "dead-end", "people who enjoy rationalizing their ignorance"... It doesn't matter that you may be right in some cases; there is no glory in being right to insult people, and it is shameful to do it wrongly.

- Avoid making preposterous claims about how a vague idea will solve all our problems.

In other words, I suggest you try to follow a rigorous scientific methodology. Concentrate on diffusing knowledge, try do to honest, factual and polite assessments of the related works (it's better to be too kind than to criticize out of ignorance), and to make reasonable claims (something doesn't need to solve all our problems to be interesting).

PS: I think you're wrong on what dynamic typing is -- and that's the most severe flaw I found in this post -- but maybe that's the topic for a different comment of mine or post of yours. But I can't resist giving the core idea: I think there is a continuum between runtime test/check and static proof, and static proofs is in general more costly than test, for a greater benefit (the cost difference is higher for most complex guarantees, on a continuum from "memory safety" to "absolute total correctness including resource usage etc."). What you want is to let people choose at which point in this continuum there want to be, and move along the line easily, according to their priorities and constraints. Most dynamic languages are very bad at this (no static proof possible), but static languages do not fare so well in practice (haskell : unusable for strong specification, impractical for very dynamic checks; agda : impractical for dynamic checking). Racket (with the ubiquitous use of contracts, plus the static Typed Racket option) is a serious contender (as, say, Haskell) along with current researchy practical dependently typed languages.

January 1, 2012 12:47 PM

Anonymous said...

There are a few fundamental problems with your vision.

* assembly is imperative and has side effects. ie, the interface to the computer is not functional.

* most coders haven't even heard of Haskell.

* Type systems are a grief to deal with formally until your program is "big enough".

* code-in-database systems keep getting created and ignored. I think there's a reason for that...

I see the future becoming functional, but with optional type systems

January 1, 2012 10:11 PM

Anonymous said...

gasche: your analysis is spot on, do you have a blog? Would love to read more from you.

The arrogance shown in Paul's post is sadly widespread in the Scala community, although I think Paul is one of the good guys, it's just that some of the more arrogant folks have probably dyed off a bit on him. Just like you, I would love to read more thoughts from Paul without the condescending attitude, I'm sure I would learn a lot.

Personally, I find close to no evidence that FP is at a tipping point, it's just that some of its more practical ideas are slowly trickling into the mainstream, and that's great for all of us.

January 2, 2012 8:37 AM

Anonymous said...

In "The future of programming" You'll just tell Mr Watson how you want your computer to behave and what to compute. And he'll make it so.

The Mr. Watson I'm referring to is:
http://www-03.ibm.com/innovation/us/watson/index.html

Bang! And, we are all out of business. Most programming is reduced to the same as playing and games like chess or ludo.

January 3, 2012 1:33 AM

khinsen said...

Thanks for this interesting post, which addresses many issues that software developers face today. You don't say anything about the tine scale on which you expect your predictions to come true. Personally, I think it's closer to a century than a decade.

There is one specific point I disagree with, though:

"Today, many of the required techniques are known, and expressing even the most imperative-seeming program functionally is possible, and for an experienced functional programmer, pretty effortless. "

The situation has certainly improved compared to ten years ago, but we are not there yet. My own field of work, scientific computing, should in theory be the first to adopt FP because our "business logic", which is actually mathematics, fits FP perfectly well. But when it comes to handling big data efficiently, FP still fails miserably. Ugly old Fortran code lets me invert a 1 GB matrix in-place. With today's FP languages, I can consider myself lucky if I can get my inverse

using less than 10 GB of memory. Of course everyone is welcome to prove me wrong!

January 3, 2012 8:44 PM

Paul Chiusano said...

@khinsen - I'm not really sure about timeframe and although I tossed out 10-20 years, I don't want to make predictions about that. :) I also don't want to make it sound like all these things I'm proposing are inevitable - maybe it will turn out the things I'm proposing are just not good ideas, are impossible, unworkable, or will become irrelevant in response to other technological developments!

If any of this stuff is going to happen, though, 100 years is an awfully long time. Computing might be completely unrecognizable then, we might have strong AI, and programming might be a "solved problem"!

Regarding your point about performance, mutation is totally fine in FP, you just have to do it in a way that preserves referential transparency, which is not hard. Functional programmers do this sort of thing all the time - see the ST monad in Haskell. In Scala we don't always use the ST monad, but it's quite common to write some (unobservable) imperative code in the implementation of a pure function.

I think there is a common misconception that FP means "no mutation". FP is not so much a restriction on what programs you can express, only a restriction on *how* you express them. Mutation and other effects are fine in FP, as long as they don't break referential transparency. You'll sometimes hear people talk about a "functional implementation" (where the function implementations don't even use mutation or unobservable side-effects locally) vs just a "functional API" (which might use imperative constructs internally).

At a cost of some efficiency (which may be unacceptable in some cases), functional implementations are often much easier to write and get correct. They can also be more efficient in some cases (due to greater sharing, and avoiding need to pessimistically copy data), and usually have a better API (due to not requiring use of tricks like ST to ensure side-effects are not observable).

January 3, 2012 9:39 PM

khinsen said...

I do realize that mutation is possible in FP, and I agree that *in principle* it should be possible to write things like in-place matrix inversion using suitable frameworks such as Haskell's ST monad. But I haven't seen it done, and it's not obvious either how to do it.

Haskell's ST monad would need an array-valued variant of STRef with an efficient implementation of access to individual elements to get the work done. Perhaps this is simple to do and just needs to be done. Perhaps this requires some serious preparation work on the Haskell code generator. Perhaps there are even some more fundamental problems to be solved first. I can't say. All I *can* say is that right now, I couldn't write my applications in Haskell at any reasonable level of efficiency.

January 4, 2012 12:21 AM

Paul Chiusano said...

@khinsen - I am not really an expert on writing low-level Haskell code but I suspect you can get good performance by not being so fine-grained with how you bind to the imperative code - so don't have each array cell update be an ST action, then try to write a high performance array or matrix algorithm entirely in the ST monad. Instead, write a block of imperative code in C for doing the matrix inversion and whatever else, then use the FFI to give it a nice API (I actually wonder if this has already been done for matrices). People in #haskell on freenode or on the mailing list would probably be of more help.

In Scala, I would basically do the same thing, and write a block of imperative Scala that looked a lot like C and ran as fast as possible. I don't think Haskell has the same convenient access to a high performance strict imperative language from within Haskell, but that is partially what the FFI is for.

Some people may disagree with me, but I don't think it is worth spending a ton of time trying to coax the Haskell compiler into generating good code for imperative, low-level Haskell. Yeah, it may be possible if you have a really good understanding of how ghc compiles code and are willing to play around with it for a while, but why bother when you can write actual imperative code in C and access it from Haskell with very little boilerplate?

January 4, 2012 4:41 AM

khinsen said...

Your reply actually illustrates my point: today's FP languages can't handle certain tasks involving large data sets efficiently. If the future of programming is supposed to be defined by lazy (and thus pure) FP, then going back to C for a frequent mathematical algorithm is not an acceptable option.

Of course I hope FP research will address such issues and find solutions, but until I see it happen I don't expect to convince many of my colleagues of the utility of FP. It's not that I don't try (see this article, for example), but I know when to stop :-)

January 4, 2012 11:55 PM

Paul Chiusano said...

@khinsen - FP is about preserving referential transparency in your programs. It does not matter if every subexpression in a (pure) function is RT. Imperative implementations are fine, and probably necessary for certain algorithms. IMO it would be a mistake to infer from this that FP is not worth it, and we should all just continue imperative programming in imperative languages as usual - that would be throwing the baby out with the bathwater. The benefits of preserving RT throughout your code where possible are enormous -- if you cannot convince your colleagues of the utility of FP because some functions seem to require imperative implementations, then you are not doing a very good job. :)

Now, I do agree that the interface between imperative and functional code could be improved. It is not ideal to have to call out to C, or to the imperative portion of Scala, and get no help from the compiler that local violations of RT have not escaped the intended scope (though I do think that even the difficulty here is somewhat overblown - I have written imperative code accessed from

pure Scala, and I find it is pretty easy to mentally certify side-effects do not escape their intended scope). But in the future, and I don't think this is too far off, I suspect we will have effect systems of some sort that can be used to track/verify the scope of side-effects. Functional code will then have access to a strict, high performance imperative language, tranparency accessible, and the effect system will track the scope of imperative updates. Again, the fact that these features are still being researched does not mean we shouldn't use existing functional languages like Haskell, which still IMO offer huge benefits over existing mainstream languages.

Also, as an aside, something I have started to notice from talking to programmers is they often drastically overestimate the portion of their code that needs to (locally) break RT for their programs to run efficiently. There are myriad of techniques that exist for factoring code and effects and more are being discovered by functional programmers all the time. For a well-factored codebase, there will be an awful lot of pure code, probably the vast majority in many cases.

PS - the link to your article is currently not working due to "servers maintenance". :(

January 5, 2012 6:46 AM
PJW said...

"Dynamic typing will come to be perceived as a quaint, bizarre evolutionary dead-end in the history of programming."

Harsh, maybe overly so. I think a better analogy would be a useful but leaky abstraction. The abstraction will be built anew with out the leaks, new shine paint and a motor to boot.

January 6, 2012 5:13 PM
Shipping News said...

Marine and offshore services firm Penguin International has commenced legal proceedings against Wartsila Ship Design Norway in connection with a contract termination. Maritime Security

January 7, 2012 3:36 AM
Andreas Scheinert said...

@khinsen Simon Peyton jones claims Matrix stuff should be fine now. Dont know though which library exactly he is talking about. Source: http://channel9.msdn.com/Blogs/Charles/YOW-2011-Simon-Peyton-Jones-and-John-Hughes-Its-Raining-Haskell

regards andreas

January 7, 2012 1:27 PM
khinsen said...

The people I have a hard time convincing are those whose code is 80% in-place array manipulation for efficiency reasons.

Of course I have a hard time convincing others as well, because of the usual inertia effect and lack of motivation to learn something new, but in those cases I am more optimistic :-)

Sorry about the server maintenance, IEEE chose the wrong moment for that ;-)

January 8, 2012 1:45 AM

Post a Comment

Older Post Home

About Me

Paul Chiusano
Boston, MA, United States

View my complete profile

We're hiring

As of February 2011, my company is looking for people with a strong background in functional programming. Contact me if interested: paul.chiusano@gmail.com

Labels

- economics (5)
- haskell (20)
- health (1)
- mathematics (1)
- politics (1)
- programming (26)
- scala (16)

Twitter updates